Supporting Behavior-based Architectures on Small Processors using Guarded Commands

Edward C. Epp, Ph.D.
Intel Research
Email: edward.epp@intel.com

Abstract – *Distributing robotic sensing and control to remote units often simplifies design, increases modularity, improves electromechanical characteristics, and promotes reuse. Distributing intelligence to these remote units is a challenge because of constrained memory (a few kilobytes). This paper describes the Guard Based Language (GBL) that is a C-like language using Dijkstra's guarded extensions. Guarded extensions provide a declarative feel to the language making it easier to map intelligence to code. The resulting compiled programs fit into small memory footprints and have little execution overhead. This paper begins with a brief introduction to Dijkstra's guards. This is followed by the description of GBL. Examples are provided of GBL programs and how they are translated to make them run efficiently on small systems. The impact of testing and modifying GBL programs is briefly addressed. Finally, open questions surrounding GBL are examined.*

1 Introduction

One of the many design tradeoffs one makes when constructing an autonomous robot is its software architecture. For example, on the software side one may choose behavior control [1] or sense-plan-execute [2]. On the hardware side one may choose a centralized control processor or distributed control/sensing processors.

This paper assumes distributed control processors. We will focus on distributed nodes that are memory-constrained (a few kilobytes). For example, Rocky 8, a JPL rover test-bed, uses Remote Engineering Units (REUs) that implement subsets of the control/sensing system design [3]. REUs simplify design, increase modularity, reduce wire length, reduce wire count, reduce thermal loss, reduce mass, reduce noise, are reusable, and accommodate design changes [4].

In addition to distributing control and sensing, one may also distribute intelligence. For example, programming primitive behavior into a remote node allows a node to respond quickly to threats or opportunities without being delayed by a centralized processor. This is not unlike nodes in the spinal cord that allow us to respond quickly to pain. However, distributing intelligence is a challenge. Because of memory constraints, it is difficult to utilize programming abstractions that aid in our effort to express reasoning [5]. It is difficult to forgo these abstractions because we know that the shorter the distance from idea to code, the easier it is to understand, test, and modify our ideas.

Traditional imperative languages, such as C, can present a challenge for representing behavioral architectures. For example, processes are often used to encapsulate behaviors. However, the mapping is not clear between a behavioral architecture and C procedures and processes. In addition, given constrained memory, processes may be difficult to support.

This paper documents an experiment with a language based on C that contains guarded commands to more naturally express behaviors with little memory or processor overhead.

2 Guards

Dijkstra was interested in establishing a "formal calculus" to derive correct programs [6]. He introduced guarded commands because their nondeterministic nature simplified program derivation. Dijkstra illustrated the semantics of guards with the following example:

```
q1,q2,q3,q4 := Q1,Q2,Q3,Q4;
do q1 > q2 ∅ q1,q2 := q2,q1;
 □ q2 > q3 ∅ q2,q3 := q3,q2;
 □ q3 > q4 ∅ q3,q4 := q4,q3;
od.
```

This code fragment sorts 4 values such that `q1` receives the smallest value, `q2` the next highest, `q3` the next, and `q4` the highest value. The first line is an assignment statement with multiple variables on the left-hand side and multiple expressions on the right-hand side of the assignment operator; `q1` gets the value `Q1`, `q2` gets the value `Q2`, and so forth. This notation is convenient because it allows a variable swap with a single assignment statement, i.e., `q1,q2 := q2,q1`. The iteration construct is enclosed with the **do** ... **od** pair. Execution remains within the loop as long as at least one of its

guards is true, for example `q1 > q2`. The order of guard evaluation is nondeterministic. When all guards are false, control leaves the loop. The `[]` is used to express nondeterminism and thus separates guarded commands.

Since Dijkstra's introduction of guards, several languages have incorporated them. Two languages of particular interest can be found in Hansen [7] and in Ishikawa, Tokuda, and Mercer[8].

3 Guard Based Language (GBL)

The author worked with Keith Rule and Tim Sauerwein of Tektronix to develop a Guard Based Language (GBL.) C was a natural base because of its heavy use for embedded systems at Tektronix and its simplicity. GBL's purpose was to create a framework on which we could evaluate guard-based languages.

A subset of GBL syntax follows and illustrates the key additions to C.

```
<program>     ::= <declaration> {<declaration>}
<declaration> ::= <variable declaration> |
                  <function declaration> |
                  <guarded command>
<guarded command> ::= when (<guard>) <priority>
                      <guard body>
<guard>       ::= <boolean expression>
<priority>    ::= priority <integer constant>
<guard body>  ::= <block>
```

Dijkstra's example would be written as follows in GBL:

```
void Init()
{
  q1 = Q1;
  q2 = Q2;
  q3 = Q3;
  q4 = Q4;
}

void Swap(int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}

when (q1 > q2) priority 0
{
  swap(q1,q2);
}

when (q2 > q3) priority 0
{
  swap(q2, q3);
}

when (q3 > q4) priority 0
{
  swap(q3,q4)
}
```

Obviously, the above example does not represent an efficient means for sorting 4 values. Its utility is because it is a simple example.

To understand how this program works, it is necessary to begin with GBL's run-time system. When a GBL program starts up, the first thing that occurs is the initialization of global variables. This is followed by a call to the `Init` function if it exists. The `Init` function is responsible for putting the embedded system into a predefined start state. This may involve setting register values, resetting peripherals, or taking initial environmental readings. Thus, the `Init` function has many of the characteristics of `main` in C. However, when the `Init` function terminates, the program is not terminated, as it would in C. Instead, the program enters its main operation mode. Within this mode, the run-time system continually poles each guard. When a true guard is found, its body is executed. GBL forever executes guard bodies whose guards are true. It does not halt when all guards are true.

Dijkstra places no limit on the number of loops with guarded commands. Each loop is stated explicitly. In GBL there is only one implied loop, which includes guarded commands. All guarded commands are embedded within this implied loop. Explicate loops are allowed, but they may not enclose guarded commands. These restrictions are for simplicity of the compiler and the code it generates.

Notice that GBL guards are evaluated deterministically. A priority clause has been added to allow the programmer control over the order in which guards are evaluated. This determinism is used to allow a given behavior to subsume another behavior.

4 Program Correctness

In a more traditional C approach to programming behaviors, a process is assigned to each behavior, plus additional processes are created to handle behavior arbitration and motor control. See Listing 3. Processes require significant memory and CPU resources and their underlying runtime schedulers are surprisingly difficulty to write correctly.

In addition, testing concurrent systems is problematic because the programmer has to account for all possible interleaving of concurrent execution. One of the problems shared by many testing methodologies is the combinatoric problem that arises in large systems. The sheer number of unique test cases becomes difficult to manage. This problem is particularly prevalent in concurrent systems. For example, Taylor, Levine, and Kelly [9] describe how techniques used in sequential programming are difficult to apply to concurrent programs. The scheduler in concurrent languages presents a key difficulty. A program tested under identical input data may exhibit markedly different behavior. Taylor, et.al. solved this problem by ignoring the effect of the scheduler and confined their testing to state transitions that have a great potential for errors. They concentrate heavily on *all-possible-*

rendezvous criterion (using the Ada model for concurrency). Limiting the number of task transitions helps manage the number of test cases.

Guarded commands are an attractive alternative to concurrency. Once execution enters a guard body, it may not be interrupted. It is assumed that guard bodies are short code fragments that evaluate quickly. This constraint simplifies the GBL runtime. Disallowing interruptions within guard bodies makes it much easier to reason about and test the program.

Exceptions are made for certain hardware and software interrupts. They may interrupt the execution of guard bodies but restrictions are placed on interrupt handlers to maintain ease of analysis. Although an interrupt may occur in the middle of executing a guard or guard body, its execution is not allowed to have any side effects that influence the execution of a guard or its body. For example, suppose the embedded system has an interrupt driven timer. The timer may be implemented with an interrupt that updates a counter. To simplify interactions, the interrupt handler may not change the time. For example,

```
int timerUpdateRequest = 0;
int currentTime = 0;

void timerInterrupthandler()
{
  timerUpdateRequest = 1;
}

when (timerUpdateRequest) priority 0
{
  currentTime++;
  timerUpdateRequest = 0;
}

when (...) ...
{
  ...
  if (currentTime < 10)
  ...
  // location x
  ...
  if (currentTime < 10)
  ...
}
```

In the code fragment above one does not have to account for unexpected changes in the current time at "location x." The interrupt handler is not allowed to change the current time because it would affect the execution of a guard body, therefore, the interrupt handler changes the timerUpdateRequest flag. The timerUpdateRequest flag is allowed only to appear by itself within one guard.

5 Example GBL Programs

The author happened to have several 68HC11 ™ controlled robots. As a result, GBL was written to generate code for the 68HC11. Testing began on small systems. Each robot had 512 bytes of EEPROM with 256 bytes of RAM.

What follows is a GBL program fragment that instructs the robot to seek dark. The robot has two light sensitive “eyes” that direct the robot toward the shadows. It also has a bumper that allows the robot to respond to obstacles. The `Init` and movement functions have been removed for simplicity.

```
// When right bumper hit, backup and turn left
when ((bumper & 0x80) == 0) priority 0
{
  reverse();
  wait(10);
  left();
  wait(5);
}

// When left bumper hit, backup and turn right
when ((bumper & 0x40) == 0) priority 0
{
  reverse();
  wait(10);
  right();
  wait(5);
}

// When right eye is darker than left, go right
when (rightEye < leftEye &&
      leftEye – rightEye > 0x10) priority 2
{
  right();
}

// When left eye is darker than right, go left.
when (leftEye < rightEye &&
      rightEye - leftEye > 0x10) priority 2
{
  left();
}

// At all other times go forward
when (1) priority 9
{
  forward();
}
```

Listing 1 – Dark seeking program in GBL

The code in Listing 1 is easy to understand and when compiled with all of its movement and timing routines, requires less than 512 bytes.

Using Dijkstra's guarded commands the above program would be organized as follows (much of the detail is removed for brevity):

```
do right bumper hit     ∅ backup and turn left;
□  left  bumper hit     ∅ backup and turn right;
□  leftEye > rightEye   ∅ turn right;
□  leftEye < rightEye   ∅ turn left;
□  not anything above   ∅ go forward;
od.
```

The last guarded command assures that at least one guard is true, so the loop will never exit.

An important feature of GBL is that it is easy to add behaviors. For example, adding the rest (stop) behavior when a robot finds a dark corner can be implemented by inserting the following code into Listing 1.

```
// When a dark spot is found, stop.
when (rightEye < 0xA0 && leftEye < 0xA0)
{
  stop();
}
```

However, caution is in order. It is trivial to create programs in which guards starve others. For example, in the traffic light snippet below, one of the guards will always be true. Thus, any lower priority guards will never trigger. One hopes static analysis may be of some assistance in finding some of these problems.

```
when (east == green) priority 5
{
  east  = red;
  north = green;
}
when (north == green) priority 5
{
  north = red;
  east  = green;
}
```

6 Implementation

The GBL compiler translates a program, such as the one in Listing 2, into an equivalent assembler program that implements the finite state machine in Figure 1.

```
int option  @ 0x1039; // 68HC11 option register
int adctl   @ 0x1030 = 0x30; // set A/D control
int leftEye @ 0x1031; //ADR1 maps to bit 0 porte

void Init()
{
  option |= 0x80;            // turn A/D on
}

// Guard 1: When bumper is hit, back up & turn.
when ((bumper & 0x40) == 0) priority 0
{
  reverse(); wait(10);
  left();    wait(10);
}

// Guard 2: When a dark spot is found, stop.
when (rightEye < 0xA0 &&
                    leftEye < 0xA0) priority 1
{
  stop();
}

// Guard 3: When the right eye is darker than
the left, go right.
when (rightEye < leftEye &&
         leftEye - rightEye > 0x10) priority 2
{
  right();
}

// Guard 4: When the left eye is darker than the
// right, go left.
when (leftEye < rightEye &&
         rightEye - leftEye > 0x10) priority 2
{
  left();
}
```

Listing 2 – Example program

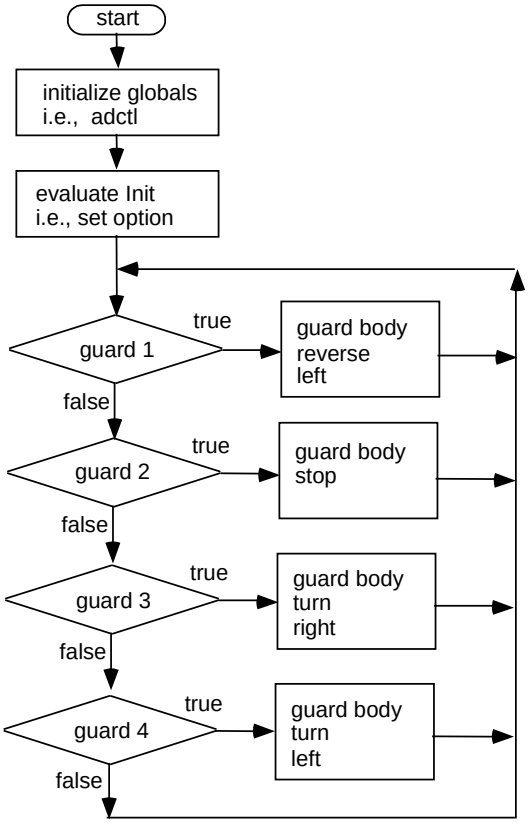

Figure 1 – Finite state machine for Listing 2

It is easy to translate the above finite state machine into a small footprint. Guard priority is achieved by ordering the guards. The higher their priority, the nearer the top of the chain they appear. (Priority 0 is the highest followed by increasingly positive integers.)

Because there is not enough space to support a real-time operating system, the GBL generated code does not rely on any operating system calls. The GBL compiler generates code for the guard scheduler and all IO drivers.

7 Example using Interactive C

For comparison, Listing 3 re-implements Listing 1 using processes. Jones and Flynn[10] provide the inspiration for this approach.

The program in Listing 3 is complex because the code that implements each behavior is distributed. Flags are used to tie the behaviors together. Code distribution also makes the program more difficult to modify. Adding the "rest" behavior requires modifications to five code locations. See Listing 4. The author found himself making multiple errors of omission when he added behaviors.

```
/* set if a particular behavior is triggered */
int cruiseOutputFlag    = 0;
int darkRightOutputFlag = 0;
int darkLeftOutputFlag  = 0;
int bumpRightOutputFlag = 0;
int bumpLeftOutputFlag  = 0;

/* possible motor commands */
int CRUISE     = 10;
int DARKLEFT   = 12;
int DARKRIGHT  = 13;
int BUMPLEFT   = 14;
int BUMPRIGHT  = 15;

int motorInput = 10;

/**** trigger cruise forward behavior *********/
/* not really needed */

void cruise()
{
  while (1) {
    cruiseOutputFlag = 1;
    defer();
  }
}

/**** trigger dark to the right behavior ******/

void darkRight()
{
  while (1) {
    if ((analog(RIGHTEYE) > analog(LEFTEYE)) &&
       ((analog(RIGHTEYE) –
                analog(LEFTEYE)) > THRESHHOLD)) {
      darkRightOutputFlag = 1;
    }
    else
      darkRightOutputFlag = 0;
    defer();
  }
}

/**** trigger dark to the left behavior *******/

void darkLeft()
{
  … similar to darkRight
}

/**** trigger bump right behavior *************/

void bumpRight()
{
  while (1) {
    if (digital(RIGHTBUMPER))
      bumpRightOutputFlag = 1;
    else
      bumpRightOutputFlag = 0;
    defer();
  }
}

/**** trigger bump left behavior *************/

void bumpLeft()
{
  … similar to bumpRight
}

/** arbitrate - select highest-level behavior */

void arbitrate()
{
  while(1) {
    if (cruiseOutputFlag == 1)
      motorInput = CRUISE;
    if (darkRightOutputFlag == 1)
      motorInput = DARKRIGHT;
    if (darkLeftOutputFlag == 1)
      motorInput = DARKLEFT;
    if (bumpRightOutputFlag == 1)
      motorInput = BUMPRIGHT;
    if (bumpLeftOutputFlag == 1)
      motorInput = BUMPLEFT;
    sleep(0.01);
    defer();
  }
}

/**** vroom - realize the behavior ***********/

void vroom()
{
  while(1) {
    if (motorInput == CRUISE) {
      motor(RIGHT,50);
      motor(LEFT ,50);
    }
    if (motorInput == BUMPRIGHT) {
      motor(RIGHT,-25);
      motor(LEFT ,-25);
      sleep(1.0);
      motor(RIGHT,100);
      motor(LEFT ,-100);
      sleep(0.5);
    }
    if (motorInput == BUMPLEFT) {
      motor(RIGHT,-25);
      motor(LEFT ,-25);
      sleep(1.0);
      motor(RIGHT,-100);
      motor(LEFT ,100);
      sleep(0.5);
    }
    if (motorInput == DARKRIGHT) {
      motor(RIGHT,0);
      motor(LEFT ,100);
    }
    if (motorInput == DARKLEFT) {
      motor(RIGHT,100);
      motor(LEFT ,0);
    }
    defer();
  }
}

/**** main - start up the processes **********/

void main()
{
  start_process(cruise());
  start_process(darkRight());
  start_process(darkLeft());
  start_process(bumpRight());
  start_process(bumpLeft());
  start_process(arbitrate());
  start_process(vroom());
}
```

Listing 3 – Dark seeking program using processes

8 Conclusion

Our experience with these robots has shown that guarded commands have great promise for writing effective embedded system controllers for small systems. We found the code easy to write, easy to reason about, and easy to evaluate.

Our current experience with GBL is still limited. There are many questions we would like to answer.

```
The following globals must be added to the
beginning of the program.

  int restOutputFlag      =  0;
  int REST                = 11;

The following task must be added.

void rest()
 {
   int threshHold = knob();
   while (1) {
     if   ((analog(RIGHTEYE) > threshHold)  &&
           (analog(LEFTEYE ) > threshHold)) {
       restOutputFlag = 1;
       }
     else
       restOutputFlag = 0;
     defer();
   }
 }

The following must be added to the arbitrate
process.

    if (restOutputFlag == 1)
      motorInput = REST;

The following must be added to the process motor
process (vroom.)

    if (motorInput == REST) {
      motor(RIGHT,0);
      motor(LEFT ,0);
    }

The following must be added to main.

  start_process(rest());
```

Listing 4 – Adding "rest" behavior to Listing 3

- It is not clear how guard based programs will scale. Current experience is based on small programs. Guards have some similarities to rules in rule base systems. Maybe GBL will suffer from some of the same problems that occur in rule-based systems as the number of rules becomes large[11].
- It is not clear what kinds of issues GBL programs may experience around deadlock, race conditions, and process starvation.
- Embedded systems have tight timing constraints. In *hard* real time systems the program must meet its timing constraints or it fails. How well do guards work in such systems? Are there extensions we can make to the language to facilitate hard real time? Can we us static analysis of the generated code to assure timing constraints are met in small programs?
- Doing proof of correctness analysis on programs can be beneficial in applications where failure is expensive (e.g., space missions). It would be interesting to determine which GBL language features are conducive to analysis. For example, what is the impact of the priority clause? It increases determinism but does it also make proof of correctness more difficult?

9 Acknowledgments

I want to acknowledge the inspiration Tim Sauerwein, David Maguire, Keith Rule, and Wendell Damm of Tektronix and undergraduate researchers Isaac Oram and Kevin Jackson-Mead of The University of Portland. This effort was supported in part by a grant from National Science Foundation (NSF RUI CCR-9407110).

10 References

[1] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation.* RA-2, (April 1986) pp.14-23.

[2] R. Simmons. Structured Control for Autonomous Robots. IEEE Transactions on Robotics and Automation, 10(1): 34-43, 1994.

3 R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, H. Das. CLARAty: Coupled Layer Architecture for Robot Autonomy. JPL Technical Report D-19975, Dec 2000.

4 D. Hykes, E. H. kopf, G. S. Bolotin, J. Waters, K. A. Mehaffey, M Bell, S. M. Park. Packages of Circuitry for Controlling a Robotic Vehicle. JPL New Technology Report NPO-20763, June 2001.

5 D. Brugali and M. E. Fayad. Distributed Computing in Robotics and Automation. IEEE Transactions on Robotics and Automation. 18(4): 409-420, August 2002.

6 E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM. 18(8): 453-457, August 1975.

7 P. B. Hansen. Distributed Processes: A Concurrent Programming Concept. Communications of the ACM. 21(11): 934-941, November 1978.

8 U. Ishikawa, H. Tokuda, and C. W. Mercer. Object-Oriented Real-Time Design: Constructs for Timing Constraints. ECOOP/OOPSLA '90 Proceedings. 289-298. October 1990.

9 R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural Testing of Concurrent Programs. IEEE Transactions on Software Engineering. 19(3): 206-215, March 1992.

10 J. L. Jones and A. M. Flynn. Mobile Robots: Inspiration to Implementation, Chapter 9: Robot Programming. A. K. Peters, 1993.

11 M. D. Rychener. Production Systems as a Programming Language for Artificial Intelligence Applications. Dissertation submitted to the Department of Computer Science at Carnegie-Mellon University. December 1976.